Sender Porting Guide

Document Number: TN-9.X.X

Document Version: 0.4

Table of Contents

List of Figures

List of Tables

Associated Documents

- NMRA Decoder Test Files
  https://app.box.com/s/7b93a06edb8f0f1cea35
  - TN9.1.1 NMRA DCC Decoder Test User Manual.pdf
  - TN9.1.2 Sender Board Theory of Operation.pdf
- Sender V2 System Document Folder
  https://app.box.com/s/pfwr5yevja99bk1fpr1so5lli8n5mgbn
  - Sender V2 Schematic.pdf
- Sender V3 System Document Folder
  https://app.box.com/s/vkkzel32wlmhs3nzktazhwrp6u5ibkke
  - TN9.1.3 Sender V3 Getting Started Guide.pdf
  - Send 3 Rev E Board Information
- SEND.EXE Software Version B.5.9.3 Information
  https://app.box.com/s/l67mk7a1evxrr9yeazq1c70ex7w2twr7

1 Introduction

The Sender V3 System is used to manually and automatically test DCC decoders. The current system is composed of these elements:

1. A Technologic Systems TS-5300 SB/104 IBM compatible Single Board Computer (SBC) running the FreeDOS operating system.
2. A version 3 Sender board that generates a DCC bit stream and a synchronized scope trace. It also queries the state of the decoder. Information on the Sender board is at this links:
    a. Theory of Operation: https://app.box.com/s/sstsk4bp0k31bl536v22wnptdv2qoube
    b. Hardware Design: https://app.box.com/s/czn61xul9wq35sx0prgvr3rl77t7kelg
3. A C++ control program called SEND.EXE that controls the user interface, test strategy, and driver interface to the version 3 Sender board. The program consists of approximately 18,000 lines of code. This document is based on version B.5.9.3 of the SEND.EXE program. The executable, source code, and doxygen generated documentation is at this link: https://app.box.com/s/l67mk7a1evxrr9yeazq1c70ex7w2twr7

There is a desire to reduce the complexity and cost of the sender system by porting the code in item 3 above to a modern System On a Chip (SOC) IC that would combine items 1 and 2 above.

This document details the interface between the upper level portion of the SEND.EXE program known as the Send App and the version 3 Sender board device driver known as the DCC Device Driver. The desire is to use as much of the existing SEND.EXE program as possible.

2 General Requirements

2.1 The New Send Hardware Shall be Protected by a Case

The test system must be protected from damage by enclosing it in a case. Leaving it open would leave it extremely vulnerable to physical damage, electro static discharge (ESD), etc.

2.1.1 The SD Card Shall be Accessible from the Host Without Opening the Case.

Requiring the case to be opened to access the SD card would render the case useless.

2.2 Minimize Changes to the Existing SEND.EXE Code

2.2.1 Maintain the Existing File Structure

2.2.2 Maintain the Class Hierarchy Wherever Possible

2.2.3 Maintain the Public Class Method Interfaces Wherever Possible

2.2.4 Comment Out Rather Than Remove Current Code When Making Changes

It is important to compare the current code to the new code during code reviews and debugging sessions. There are two ways to accomplish this:

2.2.4.1 Comment Out & Leave the Current Code Wherever Possible

2.2.4.2 When Not Possible, Leave a Comment Indicating the Current Code Replaced

This comment shall provide the class name and method implementation replaced.

2.3 The Target System Shall Be Controllable from a Host PC

2.3.1 Windows, Mac, and Linux Hosts Shall Be Fully Supported

2.3.2 The Host Shall be Able to Read and Write Data Files from the SD Card

The minimum requirement is to upload and download files using a transfer mechanism such as ZMODEM that guarantees file integrity.

A highly desirable mechanism would be to mount the SD card to the host PC. This would allow files to be uploaded and downloaded, deleted, renamed, etc., using the standard host commands. This could be accomplished by executing a dedicated file mounting program in place of the Send program.

2.3.3 Program Download and Execution Shall be Supported

The host shall be able to download and execute programs as detailed in section 3 below.

2.4 The Minimum DCC Bit Timing Resolution Shall Be 1 Microsecond

The minimum timing resolution shall be 1 microsecond. Shorter minimum timing resolution is acceptable as long as this resolution is a multiple of 1 microsecond, allowing a 1 microsecond bit time to be represented without error.

2.5 The DCC Maximum Bit Timing Error Shall Be ±100 PPM or Better

2.6 The Maximum DCC Bit Time Shall Be 12000 Microseconds or Longer

The maximum DCC bit time shall apply to all 0 bits in the buffer.

2.7 A Real Time Clock Shall Be Provided

The system must provide a real time clock that maintains the correct time whether or not the unit is powered.

2.7.1 The Accuracy of the Real Time Clock (RTC) Shall Be ±20 PPM or Better

2.7.2 There Shall Be a Way to Set the RTC from the Host

2.7.3 There Shall Be a Way to Read the RTC from the Send Program

A method must be provided to set the real time clock from the host. A method must be provided to read the date and time from the RTC to the nearest minute. Using the standard **`time()`** and **`ctime()`** methods would minimize changes to the code.

3 Program Execution and Termination

3.1 Current Program Execution and Termination

The current Send program relies on the FreeDOS operating system to load and execute .EXE programs stored on the Compact Flash (CF) card from the command line. A number of SEND programs are stored on the card. These include new versions of the program undergoing system test, and special versions created to help manufacturers debug their decoders.

Programs and test results are copied to and from the host using the ZMODEM protocol that is supported by the TS-5300. It takes about 17 seconds to download a 163 Kbyte program to the card.

When the program terminates, control is returned to the FreeDOS command line.

3.2 New Program Execution and Termination

3.2.1 The System Shall Download a New Program in 20 Seconds or Less

The minimum requirement is to download and run a single program at a time on the target hardware.

3.2.2 Running Programs from the CF Card is Optional

If programs can terminate, a method shall be provided to execute a selected program from the CF card.

4 Build Environment and Tool Chain

4.1 Current Build Environment and Tool Chain

4.1.1 Current Source Tree

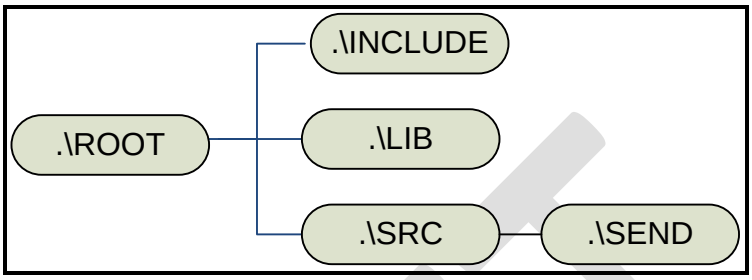

Figure 1: Current Source Tree

4.1.2 Current Tool Chain

The current build environment uses this tool chain:

1. Oracle VirtualBox version 6.0.10 virtual machine manager. This hosts the Windows XP VM that hosts the current tool chain.
2. Windows XP Ultimate SP 3 32 bit running as a virtual machine. This hosts the build tools.
3. Borland C++ 5.02. This compiles the main SEND.EXE program and the library unit test programs.
4. Borland tdstrip.exe. This deletes the symbol table from the SEND.EXE program to reduce its size.
5. Microsoft Visual Source Safe version 6.0d. This archives the source files, compares archived and edited source files, and tags builds so any release can be reproduced if needed.
6. What.exe. This program scans an executable and prints the version of each component in it based on the sccsid string.
7. DiffMerge 4.2.0. Used to regression test the new SEND.EXE versions by comparing .log files between current and new revisions using the LOG_PKTS flag to send packet output to the .log file. The 64 bit version is used on Windows 10 64 bit and the 32 bit version is used on the Windows XP VM.

4.2 New Build Environment and Tool Chain

4.2.1 New Source Tree

4.2.1.1 The New Source Tree Folders Shall Match the Current Source Tree

4.2.1.2 The New Filenames Shall Match the Current File Names Whenever Possible

4.2.1.3 The File Name Extension May Change to Match the Tool Change

4.2.2 New Tool Chain

The new build environment uses this tool chain:

5 Unit Tests

5.1 Current Unit Test Environment

5.1.1 Current Library Unit Tests

Each library class in ..\LIB has an associated unit test. The output of the unit with new library builds is compared to the output with the current build for regression testing. The unit tests and their associated library sources tested are shown below:

Table 1: Current Library Unit Tests

.\LIB Unit Test	.\INCLUDE Files Tested	.\LIB Files Tested	Description
BIT_TST.CPP BIT_TST.DSW BIT_TST.IDE	BITS.H DCC.H Z_CORE.H	BITS.CPP	Tests the **`Bits`** class that converts DCC commands into DCC bit streams.
CKTEST.CPP CKTEST.DSW CKTEST.IDE	CKSUM.H	CKSUM.CPP	Tests the **`nCheckSumFile()`** function that generates a POSIX.2 CRC on a file.
ZL_TST.CPP ZL_TST.IDE	ZLOG.H ZTYPES.H	ZLOG.CPP	Tests the **`Zlog`** class that handles status and error logging to the file system.

5.1.2 Current Send Program Unit Test

The current Send program uses these SEND.EXE unit tests:

1. Regression tests using the LOG_PKTS switch to send packet streams to the log instead of the hardware. The logs are compared to previous versions to make sure the packet streams match.
2. Scope tests to make sure the basic DCC 1 and 0 bit times are correct.

5.2 Current Send Program System Test Environment

Run automated tests with a selection of decoders with known failure patterns. The log file is compared to previous versions to make sure the results match.

5.3 New Unit Test Environment

5.3.1 New Library Unit Tests

The new library unit tests should be as close to the current library unit tests as possible accounting for changes to the tool chain, etc. Ideally, the library unit tests should be cross compiled and run on the target board. They could be compiled and run on the host although this would make the tests less accurate.

5.3.2 New Send Program Unit Test

5.4 New Send Program System Test Environment

6 Program Architecture Summary

6.1 Current Program Architecture

The current **`Z_core`** class hierarchy is below:

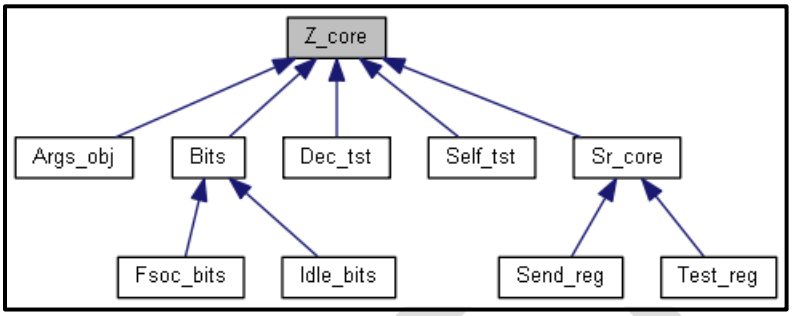

Figure 2: Z_core Class Hierarchy

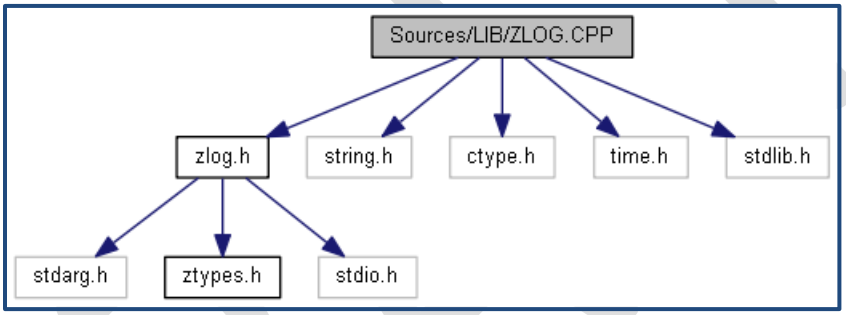

Figure 3: Zlog Class

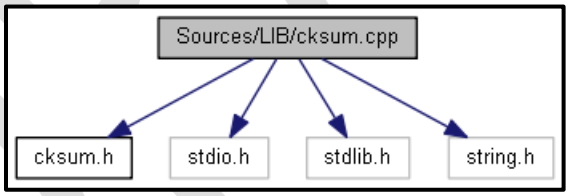

Figure 4: Cksum Module

The classes shown above are summarized below:

Table 2: Current Class Summary

Class	File	Purpose
Z_core	**Z_CORE.H** **Z_CORE.CPP**	Abstract super class for other classes. Handles object lifetime and exception handling in lieu of C++ structured exception handling.
Bits	**BITS.H** **BITS.CPP**	Library class creates DCC bit stream from DCC command primitives. The manual and automated tests use objects of this class to send commands to the Send hardware.
Fsoc_bits	**BITS.H** **BITS.CPP**	**Bits** subclass preloaded with the Fail Safe Operation command packet sequence.
Idle_bits	**BITS.H** **BITS.CPP**	**Bits** subclass to create 1 or more Idle packets.
Args_obj	**ARGS.H** **ARGS.CPP**	Parses the SEND.INI argument file and the command line arguments. Maintains and prints the resulting configuration options.
Sr_core	**SR_CORE.H** **SR_CORE.CPP**	Abstract super class modeling the Send hardware registers.
Test_reg	**TEST_REG.H** **TEST_REG.CPP**	**Sr_core** subclass that simulates the Send hardware using the test vectors.
Send_reg	**SEND_REG.H** **SEND_REG.CPP**	**Sr_core** subclass that communicates with the Send hardware.
Self_tst	**SELF_TST.H** **SELF_TST.CPP**	Runs the self test on the Send hardware.
Dec_tst	**DEC_TST.H** **DEC_TST.CPP**	Runs the automated decoder tests.
Zlog	**ZLOG.H** **ZLOG.CPP**	Library class that logs status and error data to the .log and .sum files.
Cksum	**CKSUM.H** **CKSUM.CPP**	Library module that calculates the POSIX.2 checksum on a file.

6.2 New Program Architecture

The estimated amount of change for each class is shown below:

Table 3: New Class Change Estimate

Class	Change Estimate	Purpose
Z_core	Minimal	This class sends output to **stdout**. This may require changes to go to the console.
Bits	Minimal	This class has minimal interaction with the operating system.
Fsoc_bits	Minimal	Changes should be similar to the **Bits** class.
Idle_bits	Minimal	Changes should be similar to the **Bits** class...
Args_obj	Modest	Changes may be necessary to access **SEND.INI** on the SD card. This class also reads command line switches from **stdin**. Reading command line switches may not be necessary. This class uses **stdout**.
Sr_core	Unused	This class is dependent on the Send hardware and will not be used for the new program.
Test_reg	Unused	This class is dependent on the Send hardware and will not be used for the new program.
Send_reg	Extensive	This class will require extensive implementation changes to support the new DCC output method. The public interface should be maintained wherever possible.
Self_tst	Extensive	This method will require extensive implementation changes to support the new hardware. The public interface should be maintained wherever possible.
Dec_tst	Significant	The inner loop of most automated test implementation will need to be changed to work with changes to the **Send_reg** class.
Zlog	Modest	Changes may be required to the methods that log data so that it goes to the SD card.
Cksum	Modest	The path the executable will likely need to change.

7 Class and Component Details

This section details specific classes and components that either require significant change or that are important to the operation of the **SEND.EXE** program.

7.1 Bits Library DCC Packet Creation Library

The Bits class library converts DCC commands into a Byte array that can be sent to the Send hardware. A 1 bit represents a DCC 1 bit at the given total DCC 1 bit time. A 0 bit represents a DCC 0 bit at the given total and first half DCC 0 bit times.

The packet assembly methods all return a Bits& value pointing to the current object. This allows packet assembly commands to be chained. An example is below:

```
dcc_bits.clr_in().dcc_bits.put_0s(1).put_idle_pkt().dcc_bits.done();
```

The above line clears the **Bits::dcc_bits** array, adds 1 DCC 0 bit, adds an idle packet, and closes out the packet buffer.

7.1.1 Current Bits Library Interface

7.1.1.1 Current Bits Library Interface Summary

The Send hardware is Byte oriented and accepts 1 Byte at a time. The MSB of this Byte is sent first. This process is repeated until all bits are sent. The Bits library follows this pattern.

The **Send_reg::send_pkt(Bits &ibits, const char *info)** method sends a **Bits** buffer to the Send hardware.

7.1.1.2 Current Bits Library Public Interface Details

7.1.1.2.1 Bits (u_int isize)

The constructor creates a **Bits** object with an array of **isize** Bytes. It also initializes the object to be empty. The Byte array is created on the heap so repeated construction and destruction could result in heap fragmentation.

The constructor will set a **CONSTRUCTOR_OBJ_ERR** if the Byte array cannot be allocated.

7.1.1.2.2 ~Bits ()

The destructor frees the internal Byte array.

7.1.1.2.3 Rslt_t print (void) const

This method prints the current contents of the **Bits** object to **stdout**. It will print out the error bit mask if the object has errors.

7.1.1.2.4 u_int get_isize (void) const

This method returns the internal Bytes array size in Bytes.

7.1.1.2.5 u_int get_bit_size (void) const

This method returns the current number of bits in the object. This will be 0 if no items have been added.

7.1.1.2.6 const BYTE * get_byte_array (void) const

This method returns a read-only pointer to the internal Byte array.

7.1.1.2.7 u_int get_byte_size (void) const

This method returns the current number of Bytes used. Partially used Bytes will be added to this total. This will be 0 if no items have been added

7.1.1.2.8 Rslt_t get_byte (BYTE &obyte)

This method returns each Byte in the Byte array beginning with the first Byte. Each succeeding call will return the next Byte in the Byte array. It will return an error if no Byte is available.

7.1.1.2.9 BYTE get_check () const

This method returns the DCC **`check_byte`**. The **`check_byte`** is updated as each Byte in a packet is added.

7.1.1.2.10 void rst_out (void)

This resets the location of the next available Byte returned by **`Bits::get_byte()`** to the first Byte in the array.

7.1.1.2.11 Bits & clr_in (void)

This method clears all bits in the array. It also clears any warnings and resets the flipped bit, if any.

It returns a Bits & to allow chaining.

7.1.1.2.12 Bits & put_byte (BYTE ibyte)

This method adds the Byte **`ibyte`** to the array. It updates the **`check_byte`** by exclusive oring **`ibyte`** with the current **`check_byte`**.

It returns a Bits & to allow chaining.

7.1.1.2.13 Bits & put_cmd_14 (bool forward, bool lamp, int speed)

This method puts the command Byte of a 14 speed step to the array. It updates the **`check_byte`** by exclusive oring the resultant Byte with the current **`check_byte`**.

It returns a Bits & to allow chaining.

7.1.1.2.14 Bits & put_cmd_28 (bool forward, int speed)

This method puts the command Byte of a 28 speed step to the array. It updates the **`check_byte`** by exclusive oring the resultant Byte with the current **`check_byte`**.

It returns a Bits & to allow chaining.

7.1.1.2.15 Bits & put_check (void)

This method adds the **`check_byte`** to the array. It clears the **`check_byte`** once it is added.

It returns a Bits & to allow chaining.

7.1.1.2.16 Bits & clr_check (void)

This method clears the **check_byte**.

It returns a Bits & to allow chaining.

7.1.1.2.17 Bits & put_1s (u_int count)

This method adds 0 or more DCC 1 bits to the array. No change is made to **check_byte**.

It returns a Bits & to allow chaining.

7.1.1.2.18 Bits & put_0s (u_int count)

This method adds 0 or more DCC 0 bits to the array. No change is made to **check_byte**.

It returns a Bits & to allow chaining.

7.1.1.2.19 Bits & put_fsoc (void)

This method adds a fail safe operation packet sequence to the array. It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.20 Bits & put_reset_pkt (u_int packets=1, u_int pre_bits=PRE_BITS)

This method adds **packets** count (default=1) standard reset packet(s) with **pre_bits** number of preamble bits (default=**PRE_BITS**). It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.21 Bits & put_idle_pkt (u_int packets=1, u_int pre_bits=PRE_BITS)

This method adds **packets** count (default=1) idle packet(s) with **pre_bits** number of preamble bits (default=**PRE_BITS**). It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.22 Bits & put_cmd_pkt_14 (BYTE address, bool forward, bool lamp, int speed, u_int pre_bits=PRE_BITS)

This method adds a 14 speed step packet with **pre_bits** number of preamble bits (default=**PRE_BITS**). It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.23 Bits & put_cmd_pkt_28 (BYTE address, bool forward, int speed, u_int pre_bits=PRE_BITS)

This method adds a 28 speed step packet with **pre_bits** number of preamble bits (default=**PRE_BITS**). It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.24 Bits & put_acc_pkt (u_short address, bool active, BYTE out_id, u_int pre_bits=PRE_BITS)

This method adds an accessory packet with **pre_bits** number of preamble bits (default=**PRE_BITS**). It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.25 Bits & put_sig_pkt (u_short address, BYTE aspect, u_int pre_bits=PRE_BITS)

This method adds an extended accessory signal packet with **`pre_bits`** number of preamble bits (default=**`PRE_BITS`**). It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.26 Bits & put_func_grp_pkt (u_short address, enum Func_Grps func_grp, BYTE func_bits, u_int pre_bits=PRE_BITS)

This method adds a function group packet with **`pre_bits`** number of preamble bits (default=**`PRE_BITS`**). It does not add the ending DCC 1 stop bit.

It returns a Bits & to allow chaining.

7.1.1.2.27 Bits & done (void)

This method fills any remaining bits in the last Byte of the array with DCC 0 bits and finalizes the array. This method should be called when all bits have been added to the array.

It returns a Bits & to allow chaining.

7.1.1.2.28 Rslt_t set_flip (u_int bit_pos=0)

This method flips the state of the bit at **`bit_pos`** from 0 to 1 or vice versa.

7.1.1.2.29 void clr_flip (void)

This method returns a flipped bit to its normal state.

7.1.1.2.30 Rslt_t clr_bit (u_int bit_pos)

This method sets the bit at **`bit_pos`** to 0.

7.1.1.2.31 Rslt_t truncate (u_int bit_size

This method truncates the packet by **`bit_size`** bits by removing the bits from the end of the array.

7.1.2 New Bits Library Public Interface

7.1.2.1 The New Bits Library Shall Follow the Current Class as Closely as Possible

7.2 Zlog Library Data Logging Class

The **`Zlog`** class logs all status and error messages to the **`<name>.sum`** status file and the **`<name>.log`** log file where **`<name>`** is obtained from the **`Args_obj::get_cmd_name()`** method.

7.2.1 Current Zlog Library Interface

7.2.1.1 Current Zlog Library Interface Summary

The current **`SEND.EXE`** program sends all logging information to the default **`Deflog`** object. The program also uses the macros defined in section 7.2.1.3 below.

The current program does not make use of the **`Zlog`** library **`logprint()`** or **`logdump()`** family of methods

7.2.1.2 Current Zlog Library Public Interface Details

7.2.1.2.1 Zlog (void)

The constructor initializes the class variables.

7.2.1.2.2 ~Zlog ()

The destructor closes the status and log files if not already closed.

7.2.1.2.3 const char * get_cmd_name (void) const

This method returns the root command name for the status and log files. This library method is not used for this application since the command name is obtained from the **`Args_obj::get_cmd_name()`** method.

7.2.1.2.4 Bits_t get_log_mask (void) const

This method returns the **`Bits_t`** log mask. This mask determines what debug message to print by the **`logprint()`** and **`vlogprint()`** methods.

7.2.1.2.5 Bits_t get_lib_log_mask (void) const

This method returns the **`Bits_t`** library log mask. This mask determines what debug message to print by the **`lib_logprint()`** and **`vlib_logprint()`** methods.

7.2.1.2.6 bool get_no_abort_flag (void) const

This method returns the state of the no abort flag. If **`true`**, the program will not terminate the program regardless of the **`Zlog_pri`** error severity.

7.2.1.2.7 bool get_stderr_too (void) const

This method returns the state of the **`stderr_too`** flag. If set, the log and status messages will be sent to **`stderr`**.

7.2.1.2.8 const char * get_err_pri_str (Zlog_pri ipri) const

This method returns a string with the name of the corresponding **`ipri`** value.

7.2.1.2.9 FILE * get_fp_log (void) const

This method returns the file pointer of the log file.

7.2.1.2.10 FILE * get_fp_stat (void) const

This method returns the file pointer of the status log file.

7.2.1.2.11 void set_stderr_too (bool istderr_too)

This method set the state of the **`stderr_too`** flag.

7.2.1.2.12 void set_log_mask (Bits_t mask)

This method sets the debug log mask.

7.2.1.2.13 void set_lib_log_mask (Bits_t imask)

This method sets the library debug log mask.

7.2.1.2.14 void set_no_abort_flag (bool iabort)

This method sets the state of the no_abort flag. If **`true`**, the program will not terminate the program regardless of the **`Zlog_pri`** error severity.

7.2.1.2.15 void set_cmd_name (const char *icmd)

This method sets the command name used by the error, log, and status methods. If set to a value other than **`NULL`** or **`'\0'`**, the command name will be added to all the messages.

7.2.1.2.16 Rslt_t open_log (const char *fname)

This method opens the **`<fname>`** log file.

7.2.1.2.17 Rslt_t open_stat (const char *fname)

This method opens the **`<fname>`** status file.

7.2.1.2.18 void close_log (void)

This method closes the log file.

7.2.1.2.19 void close_stat (void)

This method closes the status file.

7.2.1.2.20 void errprint (const char *func, const char *fmt,...)

This method prints a complete error message at the **`LOG_ERR`** severity.

7.2.1.2.21 void errprint (const char *func, const Zlog_pri priority, const char *fmt,...)

This method prints a complete error message at the **`priority`** severity.

7.2.1.2.22 void verrprint (const char *func, const Zlog_pri priority, const char *fmt, va_list ap)

This is the core method of the error print methods.

7.2.1.2.23 void statprint (const char *fmt,...)

This method prints a complete status message.

7.2.1.2.24 void logprint (const char *func, Bits_t mask, const char *fmt,...)

This method prints a debug message if the **`mask`** bit is set in **`log_mask`**.

7.2.1.2.25 void vlogprint (const char *func, Bits_t mask, const char *fmt, va_list ap)

This method is the core of the log print methods.

7.2.1.2.26 void lib_logprint (const char *func, Bits_t mask, const char *fmt,...)

This method prints a library debug message if the **`mask`** bit is set in **`lib_log_mask`**.

7.2.1.2.27 void vlib_logprint (const char *func, Bits_t mask, const char *fmt, va_list ap)

This method is the core of the library log print methods.

7.2.1.2.28 void logdump (const char *func, const char *fmt,...)

This method prints a dump message with 0 indent. This indent level will print the time stamp and dump identifier prior to the message.

7.2.1.2.29 void logdump (int indent, const char *func, const char *fmt,...)

This method prints a dump message at the **`indent`** indent level. An indent level greater than 0 will indent the message by the specified amount and will not begin the message with the time stamp and dump identifier.

7.2.1.2.30 void vlogdump (const char *func, const char *fmt, va_list ap)

This is the core method of the dump methods.

7.2.1.2.31 void vlogdump (int indent, const char *func, const char *fmt, va_list ap)

This is the core method of the dump method that specifies the indent level.

7.2.1.2.32 void to_dump (const char *fmt,...)

This method prints a dump message with no header information. It is used for multi-line dump messages.

7.2.1.2.33 void to_log (const char *fmt,...)

This method prints a message to the log file with no header information. It is used for multi-line log messages.

7.2.1.2.34 void to_stat (const char *fmt,...

This method prints a message to the status file with no header information. It is used to print multi-line status messages.

7.2.1.3 Current Zlog Library Macros

The **Zlog** class supports multiple error and log files by instantiating separate **Zlog** objects. Generally, only one **Zlog** object is created per application. To make this simpler, the **Zlog** code creates a default log object called **Deflog**. There are also a series of macros that send data to **Deflog**. The macros are summarized below:

Table 4: Zlog Default Log Macros

Macro	Expansion
ERRPRINT	(Deflog.errprint)
VERRPRINT	(Deflog.verrprint)
STATPRINT	(Deflog.statprint)
LOGPRINT	(Deflog.logprint)
VLOGPRINT	(Deflog.vlogprint)
LIB_LOGPRINT	(Deflog.lib_logprint)
LIB_VLOGPRINT	(Deflog.lib_vlogprint)
LOGDUMP	(Deflog.logdump)
VLOGDUMP	(Deflog.vlogdump)
TO_LOG	(Deflog.to_log)
TO_DUMP	(Deflog.to_dump)
TO_STAT	(Deflog.to_stat)

7.2.2 New Zlog Library Public Interface

7.2.2.1 The New Zlog Library Interface Shall Follow the Current Class Interface

7.2.2.2 Current Zlog Library Interface Shall be Maintained

7.3 Args_obj Program Configuration

The **Args_obj** class parses the **SEND.INI** file and the command line arguments and maintains the updated list of program configuration options for use by other objects.

The 3 sources of arguments are handled in this priority:

1. Highest: Command line arguments override lower priority arguments.
2. Middle: **SEND.INI** arguments override compile time arguments.
3. Lowest: Compile time variables initialized by the **Args_obj** constructor.

7.3.1 Current Args_obj Interface

7.3.1.1 Current Args_obj Interface Summary

The configuration variables are maintained in a single global Args_obj::Args object. This must be global so that all components can access the configuration variables.

All configuration changes are done by a call to **Args.get_args()** method. These is a get method for each configuration variable.

7.3.1.2 Current Args_obj Public Interface Details

7.3.1.2.1 Args_obj (void)

The constructor initializes all compile time configuration variables.

7.3.1.2.2 ~Args_obj ()

The destructor closes the **SEND.INI** file if it is open.

7.3.1.2.3 const char * get_cmd_name (void) const

This method returns the program command name in DOS 8.3 format. This is generally "**SEND.EXE**".

7.3.1.2.4 const char * get_ini_path (void) const

This method returns the full path to the **SEND.INI** file or "**\0**" if no **SEND.INI** is found. The **SEND.INI** file is opened by the protected **Args_obj::iniopen()** method that searches for **SEND.INI** in this order:

1. Highest: The current directory.
2. Middle: The path given in the "**SEND.INI**" environment variable.
3. Lowest: The directory of the **SEND.EXE** program

The command names follow the DOS 8.3 format and the maximum path lengths are set by the DOS limit. Both of these will need to change to meet the new environment.

7.3.1.2.5 FILE * get_ini_fp (void) const

This method returns the file point of the **SEND.INI** program or NULL is not **SEND.INI** program was found.

7.3.1.2.6 u_short get_decoder_address (void) const

This method returns the address of the decoder under test. The range od decoder addresses is dependent on the decoder type.

7.3.1.2.7 u_short get_port (void) const

This method returns the base DOS IO port used by the Send hardware.

7.3.1.2.8 Dec_types get_decoder_type (void) const

This method returns the decoder type. The decoder types are defined by the Dec_types enum.

7.3.1.2.9 bool get_crit_flag (void) const

This method returns the state of the **crit_flag**. If true, all interaction with the Send hardware is protected by critical sections.

This flag was necessary with a few early PC clones. It should not be used unless absolutely necessary since it causes the system clock to lose time. It is not needed with the TS-5300 SBC.

7.3.1.2.10 bool get_fragment_flag (void) const

This method returns the state of the **fragment_flag**. If true, the automated fragment test tests all fragment, not just the standard ones.

It is not recommended to set this flag true since the nonstandard fragments can be mistaken for good packets. All standard fragments are marked with the '*' character.

7.3.1.2.11 bool get_rep_flag (void) const

This method returns the state of the **rep_flag**. If true, the automated tests are repeated indefinitely until interrupted.

7.3.1.2.12 Bits_t get_run_mask (void) const

This method returns the state of the 32 bit **run_mask**. A 1 in a given bit position indicates that the automated test should run the associate test.

7.3.1.2.13 Bits_t get_clk_mask (void) const

This method returns the state of the 32 bit **clk_mask**. A 1 in a given bit position indicates that the automated test should use the associated DCC 0 and 1 bit times.

7.3.1.2.14 bool get_manual_flag (void) const

This method returns the state of the **manual_flag**. If true, the automated tests are not automatically started. The program runs the self tests and enters the console command processor.

7.3.1.2.15 u_int get_extra_preamble (void) const

This method returns the **extra_preamble** value. This value is added to the default of 12 preamble bits for the tests that do not explicitly define the number of preamble bits.

This value is used to debug decoders that show errors with 12 bit preambles.

7.3.1.2.16 bool get_trig_rev (void) const

This method returns the state of the **trig_rev** flag. This method is used with mobile decoders. If true, the trigger command uses ½ speed forward instead of emergency stop.

This flag is used to debug mobile decoders that do not handle an emergency top command.

7.3.1.2.17 u_int get_fill_msec (void) const

This method returns the **fill_msec** value. This value determines the time the program gives the decoder to change its output state in response to a command packet.

7.3.1.2.18 u_int get_test_repeats (void) const

This method returns the **test_repeats** value. This value is used by the later tests starting with the packet fragment test. These later tests are run 2 times each to save test time

7.3.1.2.19 bool get_print_user (void) const

This method returns the state of the **print_user** flag. If true, the program prints the user information to **S_USER.TXT** and exits.

7.3.1.2.20 bool get_log_pkts (void) const

This method returns the state of the **log_pkts** flag. If true, all packet data is sent to the log file rather than to the Send hardware. The program can be run with no Send hardware present if this flag is true.

Setting this flag generates a large amount of log data. It is used for regression testing by comparing the output of a new version of the program with the output from a previous version of the program.

7.3.1.2.21 bool get_no_abort (void) const

This method returns the state of the **no_abort** flag. If true, the program will not terminate on the first error.

This flag is mainly used to debug faulty Send hardware.

7.3.1.2.22 bool get_late_scope (void) const

This method returns the state of the **late_scope** flag. If true, the scope trigger occurs at the end of the trigger packet rather than the beginning of the trigger packet.

7.3.1.2.23 bool get_ambig_addr_same (void) const

This method returns the state of the **ambig_addr_same** flag. This flag is used by the 1 ambiguous bit and 2 ambiguous bits tests. If true, the preset packet with the ambiguous bit or bits just before the trigger packet will use the same address as the trigger packet. Normally, the preset packet with the ambiguous bit or bits is sent to an address other the decoder address.

This flag should not be set except for debugging since it violates the NMRA standards to send 2 adjacent packets to the same decoder address.

7.3.1.2.24 BYTE get_aspect_preset (void) const

This method returns the value of the **aspect_preset** value. This value is used for extended accessory signal decoders. This value must be between 0 and 31 and this value will be used for the preset phase of the test.

7.3.1.2.25 BYTE get_aspect_trigger (void) const

This method returns the value of the **aspect_trigger** value. This value is used for extended accessory signal decoders. This value must be between 0 and 31 and this value will be used for the reset and trigger phases of the test.

7.3.1.2.26 bool get_lamp_rear (void) const

This method returns the value of the **lamp_rear** flag. This method is used with function decoders. If true, the decoder direction will be set to reverse prior to sending the function group 1 commands. The default is to use the forward direction.

7.3.1.2.27 BYTE get_func_mask (void) const

This method returns the state of the 5 bit **func_mask**. This value is used with function decoders. This value is used with the function group 1 commands during the preset phase of the automated tests. All bits are set to 0 during the reset and trigger phases.

7.3.1.2.28 void usage (FILE *ofp=stderr)

This method prints out the usage message to the **ofp** file.

7.3.1.2.29 Rslt_t get_args (int argc, char **argv)

This method parses the **SEND.INI** file if available followed by any command line switches to populate the various configuration variables.

7.3.2 New Args_obj Public Interface

7.3.2.1 The New Args_obj Public Interface Shall Follow the Current Class as Closely as Possible

7.4 Send_reg DCC Driver Interface

The **Send_reg** class interfaces with the Send hardware. All interactions with the Send hardware is through the public interface methods of this Class.

7.4.1 Current Send_reg Driver Interface

7.4.1.1 Current Send_reg Driver Interface Summary

The Send hardware is Byte oriented where each 0 bit is sent using the **clk_0t** and **clk_0h** bit times and each 1 bit is sent using the **clk_1t** bit time. The **Send_reg** class handles all interaction with the Send hardware.

The **Send_reg** methods are summarized below:

7.4.1.1.1 Send Board Initialization and Control

The **init_send()** method initializes the Send board hardware.

The **set_clk()** method sets the DCC 0 bit time, the DCC 0 first half bit time, and the DCC 1 total bit time in microseconds.

The **start_clk()** and **stop_clk()** methods synchronously start and stop the 1 MHz clock that is used to send packets. The DCC 0 and 1 bit times are based on the **set_clk()** values. The Send board sends the current Byte loaded to the hardware. The under run flag is set if a new Byte is not loaded into the hardware before the current Byte is completely sent. The Send board sends repeated DCC 1 bits if no Byte is being sent.

The **set_scope()** method is used to send a scope trigger coincident with the next Byte sent by the Send board.

7.4.1.1.2 Repeated Single Byte Transmission

The **send_bytes()** method sends a single Byte 1 or more times using the **set_clk()** times.

7.4.1.1.3 Send Normal Buffer

The overloaded **send_pkt()** methods send a Bytes buffer class object or a Byte buffer of a given length using the current **set_clk()** times. It is important to note that these buffers do not normally send standard DCC packets. They often send packet fragments, packets with a bad bit, etc.

The **send_rst()** method sends a normal reset packet using the current **set_clk()** times.

The **send_hard_rst()** method sends a hard reset packet using the current **set_clk()** times.

The **send_idle()** method sends an idle packet using the current **set_clk()** times.

The **send_base()** method sends a generic baseline packet using the current **set_clk()** times.

7.4.1.1.4 Send Single Stretched 0 Byte

The **`send_stretched_byte()`** method sends a single Byte with the most significant bit sent using the single stretched DCC 0 total and first half times. The remaining bits are sent using the current **`set_clk()`** times. The most significant bit must be 0 or a fatal error will occur.

7.4.1.1.5 Send Byte with 1 Ambiguous Bit

The **`send_1_ambig_bit()`** method sends a single Byte with the most significant bit sent using the ambiguous bit DCC total and first half times. The remaining bits are sent using the current **`set_clk()`** times. The most significant bit must be 0 or a fatal error will occur. This method is used to simulate a short Railcom signal or other ambiguous bit.

7.4.1.1.6 Send Byte with 2 Ambiguous Bits

The **`send_2_ambig_bits()`** method sends a single Byte with the 2 most significant bits sent using the two ambiguous bit DCC total and first half times pairs. The remaining bits are sent using the current **`set_clk()`** times. The most significant 2 bits must be 0 or a fatal error will occur. This method is used to simulate a long Railcom signal or other ambiguous bit pair.

7.4.1.2 Current Send_reg Public Interface Details

This section details the methods of the current public **`Send_reg`** interface:

7.4.1.2.1 Send_reg(void)

The constructor initializes the member variables.

7.4.1.2.2 bool get_running (void) const

This get method returns the running state of the hardware.

7.4.1.2.3 u_long get_b_cnt (void) const

This get method returns the current total count of Bytes sent.

7.4.1.2.4 u_long get_p_cnt (void) const

This method returns the current total count of packet buffers sent.

7.4.1.2.5 u_int get_scope (void) const

The method returns the state of the scope trigger.

7.4.1.2.6 u_short get_clk0t (void) const

This method returns the total DCC clock 0 time in microseconds.

7.4.1.2.7 u_short get_clk0h (void) const

This method returns the total first half DCC clock 0 time in microseconds.

7.4.1.2.8 u_short get_clk1t (void) const

This method returns the total DCC clock 1 time in microseconds.

7.4.1.2.9 bool get_do_crit (void) const

This method returns the state of the critical section flag. If true, all commands that change the state of the send board occur in a critical section.

7.4.1.2.10 bool get_swap_0_1 (void) const

This method returns the state of the DCC bit swap flag. If true, a 0 in the Byte is treated as a DCC 1 and vice versa. This is used in DCC 1 bit duty cycle test.

7.4.1.2.11 bool get_log_pkts (void) const

This method returns the state of the log packets flag. It true, packet data is sent to the log file rather than to the Send hardware.

7.4.1.2.12 BYTE get_gen (void) const

This method returns the general input Byte.

7.4.1.2.13 void set_do_crit (bool ido_crit)

This method sets or clears the critical section flag.

7.4.1.2.14 void set_swap_0_1 (bool iswap)

This method sets or clears the DCC bit swap flag.

7.4.1.2.15 void set_log_pkts (bool ilog_pkts)

This method sets or clears the log packets flag.

7.4.1.2.16 Rslt_t init_8254 (void)

This method initializes the 8254 device on the Send hardware.

7.4.1.2.17 Rslt_t init_8255 (void)

This method initializes the 8255 device on the Send hardware.

7.4.1.2.18 Rslt_t rst_8255 (void)

This method resets the 8255 device on the Send hardware.

7.4.1.2.19 Rslt_t init_send (void)

This method initializes the Send hardware.

7.4.1.2.20 Rslt_t set_clk (u_short iclk0t, u_short iclk0h, u_short iclk1t)

This method sets the total DCC 0 time, the first half DCC 0 time, and the total DCC 1 time in microseconds.

7.4.1.2.21 Rslt_t set_pc_delay_1usec (void)

This method uses the number of clock cycles required to delay 1 microsecond in a critical section. It is used with the protected **`pc_delay_high()`** and **`pc_delay_low()`** methods.

7.4.1.2.22 void set_scope (bool scope_on)

This method arms or resets the scope trigger. When armed. The scope trigger is generated coincident with the next Byte transmitted.

7.4.1.2.23 void clr_under (void)

This method clears the under run flag that is set by the Send hardware if Bytes are not sent to the Send hardware fast enough.

7.4.1.2.24 void clr_err_cnt (void)

This method clears the protected **`err_cnt`** variable. The **`err_cnt`** variable is used to limit the number of error messages if large numbers of Send hardware errors occur.

7.4.1.2.25 Rslt_t start_clk (void)

This method synchronously starts the 1 MHz clock that is used to send packets. The DCC 0 and 1 bit times are based on the **`set_clk()`** values. The Send board sends the current Byte loaded to the hardware. The under run flag is set if a new Byte is not loaded into the hardware before the current Byte is completely sent. The Send board sends repeated DCC 1 bits if no Byte is being sent.

7.4.1.2.26 Rslt_t stop_clk (void)

This method synchronously stops the 1 MHz clock that is used to send packets.

7.4.1.2.27 Rslt_t send_rst (void)

This method sends a normal reset packet using the current **`set_clk()`** times.

7.4.1.2.28 Rslt_t send_hard_rst (void)

This method sends a hard reset packet using the current **`set_clk()`** times.

7.4.1.2.29 Rslt_t send_idle (void)

This method sends an idle packet using the current **`set_clk()`** times.

7.4.1.2.30 Rslt_t send_base (void)

This method sends a generic baseline packet using the current **`set_clk()`** times.

7.4.1.2.31 Rslt_t send_bytes (u_int icnt, BYTE ibyte, const char *info)

This method sends a single Byte 1 or more times using the current **`set_clk()`** times.

7.4.1.2.32 Rslt_t send_stretched_byte(u_short iclk0t,u_short iclk0h,BYTE ibyte,const char *into)

This method sends a single Byte with the most significant bit sent using the single stretched DCC 0 total and first half times. The remaining bits are sent using the current **`set_clk()`** times. The most significant bit must be 0 or a fatal error will occur.

7.4.1.2.33 Rslt_t send_1_ambig_bit (u_short iclk0t, u_short iclk0h, BYTE ibyte, const char *into)

This method sends a single Byte with the most significant bit sent using the ambiguous bit DCC total and first half times. The remaining bits are sent using the current **`set_clk()`** times. The most significant bit must be 0 or a fatal error will occur. This method is used to simulate a short Railcom signal or other ambiguous bit.

7.4.1.2.34 Rslt_t send_2_ambig_bits (u_short iclk0t1, u_short iclk0h1, u_short iclk0t2, u_short iclk0h2, BYTE ibyte, const char *into)

This method sends a single Byte with the 2 most significant bits sent using the two ambiguous bits DCC total and first half times pairs. The remaining bits are sent using the current **`set_clk()`** times. The most significant 2 bits must be 0 or a fatal error will occur. This method is used to simulate a long Railcom signal or other ambiguous bit pair.

7.4.1.2.35 Rslt_t send_pkt (Bits &ibits, const char *info)

This overloaded method sends a Bytes buffer class object using the current **`set_clk()`** times. It is important to note that these buffers do not normally send standard DCC packets. They often send packet fragments, packets with a bad bit, etc.

7.4.1.2.36 Rslt_t send_pkt (const BYTE *ibytes, u_int isize, const char *info)

This overloaded method sends a Byte buffer of a given length using the current **`set_clk()`** times. It is important to note that these buffers do not normally send standard DCC packets. They often send packet fragments, packets with a bad bit, etc.

7.4.1.2.37 bool errprint_ok (void)

This method increments the error count and returns true if the number of contiguous errors has not been exceeded.

7.4.1.2.38 void print_stats (void)

This method prints the number of Bytes and packets sent to stdout.

7.4.1.2.39 void rst_stats (void)

This method resets the sent Bytes and sent packets count.

7.4.1.2.40 void update (void)

This method reads all the Send hardware register values into the internal state variables of the object.

7.4.1.2.41 void up_print (bool pr_hdr_flag=false)

This method updates the Send hardware register values and then prints them out using the **`Sr_core::print()`** method.

7.4.1.2.42 void gen_print (void) const

This method prints the 4 generic input bits to stdout.

7.4.2 New Send_reg Driver Interface

7.4.2.1 The New Send_reg Driver Shall Follow the Current Methods as Closely as Possible

The following methods shall be modified as follows:

7.4.2.1.1 New Init_send() Method

The new **`init_send()`** method shall initialize the new DCC packet generation firmware.

7.4.2.1.2 New start_clk() Method

The new **`start_clk()`** method will commence sending the current packet, if available. If no packet is available, it will send either continuous DCC 1 bits or DCC idle packets using the current **`set_clk()`** values.

7.4.2.1.3 New stop_clk() Method

The new **`stop_clk()`** method will immediately stop sending packets.

7.4.2.1.4 New Scope Trace Method

The current scope trace method takes advantage of the Send hardware to set the scope trace just before the affected Byte. This method cannot use this method.

A new scope trace method shall be created that gives the bit position where the scope trace should occur.

7.4.2.1.5 New Normal Packet Methods

The exact method used depends on how fast the interface can convert the set_clk() clock information and the packet buffer data into the buffer used by the DMA system. If this can be completed before the shortest previous DMA buffer has completed, the interface can be similar to the current interface.

If this cannot be done fast enough, the reset, preset, trigger, and idle packet sequences will need to be converted prior to the test run.

7.4.2.1.6 New Single Stretched 0 and Ambiguous Bit Packet Methods

These DCC bit streams will require significant change. The current program takes advantage of the Send hardware by changing the DCC 0t and 0h times for a single DCC 0 bit or 2 adjacent bits in a data Byte before returning to the normal clock 0 times.

New **`Send_reg`** packet methods must be created for these types of special packets:

7.4.2.1.6.1 New Single Stretched 0 Packet

The new single stretched 0 packet shall accept a Bits buffer together with the location of the single stretched 0, and the stretched 0 total bit time and first half bit time. It will use the current set_clk() for all other DCC 0s and 1s.

7.4.2.1.6.2 New Single Ambiguous Bit Packet

The new single ambiguous bit packet shall accept a Bits buffer together with the location of the single ambiguous bit, and the ambiguous total bit time and first half bit time. It will use the current set_clk() for all other DCC 0s and 1s.

7.4.2.1.6.3 New Double Ambiguous Bit Packet

The new double ambiguous bit packet shall accept a Bits buffer together with the location of the 2 adjacent ambiguous bits, and the ambiguous total bit time and first half bit times for the first and second ambiguous bits. It will use the current **`set_clk()`** for all other DCC 0s and 1s.

7.5 Self_tst Program Self Test

The Self_tsts class performs the static and dynamic tests of the Send hardware each time the program starts. Any errors will terminate the program unless Args.get_no_abort() returns true. The self tests are not run if the Args.get_log_pkts() returns true.

7.5.1 Current Self_tst

7.5.1.1 Current Self_tst Summary

The current self test consists of static, vector based tests and dynamic tests of the Send hardware. The Self_tst collaboration graph is below:

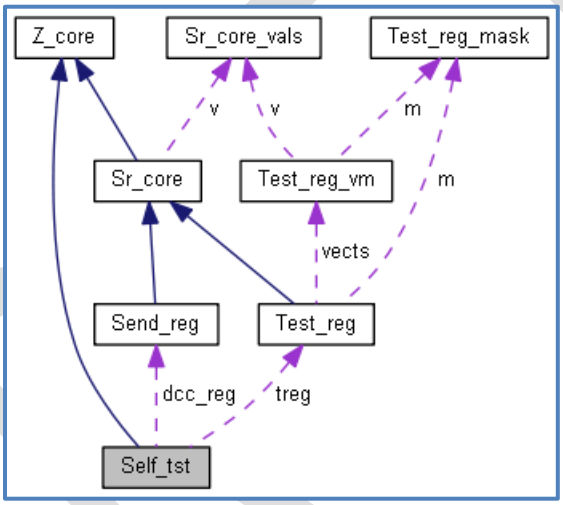

Figure 5: Self_tst Collaboration Graph

7.5.1.1.1 Current Self_tst Static Vector Tests

The current static vector tests compare the actual state variables returned from the Send hardware with a simulated Send hardware board driven primarily by the test vectors. Any difference between the real Send hardware and the simulation is flagged as an error. The TS-5300 drives the clock one clock phase at a time, comparing results with each clock step.

To minimize the numbers of test vectors when only the 82C54 counters are changing, a single test vector can be used to simulate the hardware as long as other bits do not change.

There is also an ignore mask associated with each test vector. This allows state bits to be ignored if there state is not certain yet.

7.5.1.1.2 Current Self_tst Dynamic Tests

The current dynamic test compares the number of DCC bits sent in 1 second of the TS-5300 clock. This is done to make sure the 1 MHz hardware clock on the Send hardware is operating at the desired frequency. This test verifies that the hardware clock is not mis-loaded or defective. It also tests that the TS-5300 clock is not defective.

7.5.1.2 Current Self_tst Public Interface Details

Most of the public Self_tst public methods cannot be used with the new program. The following public methods should be maintained:

7.5.1.2.1 Rslt_t test_send (void)

This method runs the static and dynamic self test of the Send hardware. It returns OK on pass or FAIL on failure. The tests abort on the first failure unless Args.get_no_abort() returns true.

7.5.1.2.2 void print_stats (void) const

This method prints the self test statistics including the number of static and dynamic tests run and the failure counts. The data is sent to the log and to the console.

7.5.2 New Self_tst

7.5.2.1 New Self_tst Summary

The new self tests will be dramatically different from the current self tests. The new self tests should follow these general criteria:

7.5.2.1.1 Self Test the General SOC Hardware

The general SOC hardware, DMA interface, etc. should be tested.

7.5.2.1.2 Test the SOC Clock if Possible

The current self test verifies that the 1 MHz clock matches the TS-5300 clock over a 1 second period. Such a test may not be possible with the new hardware.

7.5.2.1.3 Self Test the New Send Hardware Interface to the Output Pins

The new self test should verify that the DCC output and Scope output are correct by feeding these bits back to RS-422 receivers through 1K resistors. Ideally, this should be done at speed to make sure the output will work at speed.

7.5.2.2 The New Self_tst Shall Support These Public Methods

7.5.2.2.1 Rslt_t test_send (void)

This method shall run the new self tests. It shall support the Args.get_no_abort() method in the manner of the current self test. It shall support the Args.get_log_pkts() to allow unit testing without requiring the associated new interface board.

7.5.2.2.2 void print_stats (void) const

This method shall print the new self test statistics to the log file and the console.

7.6 Manual Tests

7.6.1 Current Manual Tests

The figure below shows the current manual test commands supported by the SEND.EXE program. These commands are used to set up and verify the decoder connection prior to running the automated baseline test. They are also used to debug the program and repeatedly run the self tests.

```
  Command mode (h for help, q to quit) >>
  ESC - Return to command line       h - Print header
    c - Send single clock phase      C - Send series of clock phases
    u - Clear underflow              0 - Send zeros
    1 - Send ones                    a - Send scope A pattern
    b - Send scope B pattern         o - Send scope timing packet
    w - Send warble packets          S - Send stretched 0 pattern
    r - Send DCC reset packets       d - Send DCC packets
    D - Send stretched DCC packets   s - Change loco speed, acc. output
    e - Set speed to E-STOP          f - Change loco direction, acc. on/off
    E - Set speed to E_STOP(I)       t - Run self tests repeatedly
    z - Run decoder tests            i - Send DCC idle packets
    R - Send hard resets             g - Test generic I/O
    q - Quit program
       D          S  C
       C S        C  P        O             O             O
       CSH8D CU OAO IUCB      U             U             U
       OCO2CDLN BCPCNCPD      TN            TN            TN
       UOLGCCKD FKELTLUR      PU     B      PU     B      PU     B
       TPD0QC1E AANRRKES      ULRRMMMC      ULRRMMMC      ULRRMMMC
    PA DELHH0LR LLLLALNT    0T TL10210D   0H TL10210D   1T TL10210D GEN
  ------------------------------------------------------------------------
  0x00 00000000 00000000     0 00000000     0 00000000     0 00000000 0000
  Command mode (h for help, q to quit) >>
```

Figure 6: Current SEND.EXE Manual Tests

The following subset of the above manual test commands Shall be supported by the new system.

7.6.2 New Manual Tests

The following manual test commands should be supported by the new send program.

7.6.2.1 h – Print Help Message

7.6.2.2 t – Repeatedly Run the Self Tests

7.6.2.3 0 – Send DCC 0 Pattern

7.6.2.4 1 – Send DCC 1 Pattern

7.6.2.5 S – Send Stretched 0 Pattern

7.6.2.6 i – Send Repeated DCC Idle Packets

7.6.2.7 a – Send Repeated Scope A Patterns

7.6.2.8 b – Send Repeated Scope B Patterns

7.6.2.9 r – Send Repeated Normal Reset Packets

7.6.2.10 R – Send Repeated Hard Reset Packets

7.6.2.11 d, D – Send Repeated DCC Command Packets

This manual test command sends a repeated sequence of 1 command packet followed by 1 Idle packet. The command packet sent is based on the decoder type defined by **`Args_obj::get_decoder_type()`**. The **d** command sends DCC commands with nominal DCC 0 bit times without 0 stretching. The **D** command sends a command packet with the first 0 after the first preamble stretched to the maximum positive value.

The following subcommands that work with the **d** and **D** command Shall be supported:

7.6.2.11.1 s, e, E – Change Mobile Decoder Speed or Accessory Output

For mobile (locomotive) decoders, the **s** subcommand changes the speed one step up until speed step 28 is reached and then decreases the speed one step until speed 0 is reached. The **e** subcommand sends emergency stop commands and the **E** subcommand sends alternate emergency stop commands.

For function decoders, the **s**, **e**, and **E** subcommands are not used.

For accessory decoders, the s subcommand changes the accessory output from 0 to 7 and back to 0.

7.6.2.11.2 f – Change Direction or On/Off State

For mobile (locomotive) decoders, the f subcommand changes the locomotive direction.

For function decoders, the **f** subcommand switches the 5 function group 1 outputs between 0x00 and the value defined by **`Args_obj::get_func_mask()`**.

For accessory decoders, the **f** command turns the current accessory decoder output state on and off.

7.6.2.12 z – Run the Automated Baseline Tests

This command starts the automated baseline tests. The following subcommand is active when the automated baseline tests are running:

7.6.2.13 <ESC>+q – Interrupt the Automated Baseline Tests

Pressing the **<ESC>** key immediately followed by the **q** key interrupts the automated baseline tests and returns to manual mode.

7.6.2.14 q – End the Send Program and Return to the Operating System

The **q** command exits the send program after writing out and closing the log files. This command may not be needed if no operating system is used.

7.7 Dec_tst Automated Baseline Tests

An object of this class runs the automated decoder tests and logs the results.

7.7.1 Current Dec_tst Interface

7.7.1.1 Current Dec_tst Interface Summary

7.7.1.2 Current Dec_tst Public Interface Details

7.7.1.2.1 Dec_tst (void)

This constructor initializes the member variables that must have defaults.

7.7.1.2.2 ~Dec_tst ()

This destructor is empty.

7.7.1.2.3 void print_user_docs (FILE *ofp)

This method prints the user docs including the list of tests and their associated bit mask and the list of clocks and their associated bit mask to the **`S_USER.TXT`** file.

7.7.1.2.4 void set_run_mask (Bits_t irun_mask)

This method copies the **`Args.get_run_mask()`** variable into the local **`run_mask`** member variable. This is done to speed up the access to this variable.

7.7.1.2.5 void set_clk_mask (Bits_t iclk_mask)

This method copies the **`Args.get_clk_mask()`** variable into the local **`clk_mask`** member variable. This is done to speed up the access to this variable.

7.7.1.2.6 void set_trig_rev (bool itrig_rev=false)

This method copies the **`Args.get_trig_rev()`** variable into the local **`trig_rev`** member variable. This is done to speed up the access to this variable.

7.7.1.2.7 void set_fill_msec (u_long ifill_msec=MSEC_PER_SEC)

This method copies the **`Args.get_fill_msec()`** variable into the local **`fill_msec`** member variable. This is done to speed up the access to this variable.

7.7.1.2.8 Rslt_t decoder_test (Rslt_t &tst_rslt)

This method runs the automated tests given in **`run_mask`** and **`clk_mask`** and logs the results.

7.7.2 Current Dec_tst Protected Implementation

The protected implementation is the critical part of the automated tests. The amount of program change will vary greatly based on how well the **`Send_reg`** interface as well as other class interfaces can be maintained.

7.7.2.1 Current Protected Implementation Summary

7.7.2.2 Current Protected Implementation Details

7.7.2.2.1 bool get_test_break ()

This method reads characters from the console as the automated tests are running. It searches for the '<ESC>' character followed by the 'q' character. This sequence will interrupt the automated tests and return control to the command processor.

Each test implementation calls this method at the end of each test cycle to interrupt the program and return control to the command processor. This is problematic because some test cycles take several seconds to complete.

7.7.2.2.2 Rslt_t decoder_ramp (Rslt_t &tst_rslt)

This method performs the ramp test for mobile decoders at the current overall DCC 0 and 1 bit times.

7.7.2.2.3 Rslt_t acc_ramp (Rslt_t &tst_rslt)

This method performs the ramp test for accessory decoders at the current overall DCC 0 and 1 bit times.

7.7.2.2.4 Rslt_t func_ramp (Rslt_t &tst_rslt)

This method performs the ramp test for function decoders at the current overall DCC 0 and 1 bit times.

7.7.2.2.5 Rslt_t sig_ramp (Rslt_t &tst_rslt)

This method performs the ramp test for extended accessory signal decoders at the current overall DCC 0 and 1 bit times.

7.7.2.2.6 Rslt_t decoder_ames (Rslt_t &tst_rslt, u_int pre_cnt, u_int idle_cnt)

This method performs the packet acceptance test at the current overall DCC 0 and 1 bit times.

7.7.2.2.7 Rslt_t decoder_bad_addr (Rslt_t &tst_rslt)

This method performs the wrong address test at the current overall DCC 0 and 1 bit times.

7.7.2.2.8 Rslt_t decoder_bad_bit (Rslt_t &tst_rslt)

This method performs the single bad bit test at the current overall DCC 0 and 1 bit times.

7.7.2.2.9 Rslt_t decoder_margin_1 (void)

This test does a binary search to determine the minimum and maximum DCC 1 bit times the decoder will accept.

7.7.2.2.10 Rslt_t decoder_duty_1 (void)

This tests does a binary search to determine the minimum and maximum duty cycle the decoder will accept.

7.7.2.2.11 Rslt_t quick_ames (u_int &f_cnt, const char *tst_name, u_short tclk0t, u_short tclk0h, u_short tclk1t, u_int margin_pre, bool swap_0_1)

This method is a variation of the **`decoder_ames()`** method used by the **`margin_1()`** and **`duty_1()`** methods. It can use special DCC 0 and 1 bit times. It can also swap the meaning of the DCC 0 and 1 bit if **`swap_0_1`** is true.

This method returns on the first failure or when 100 tests have passed successfully.

7.7.2.2.12 Rslt_t decoder_truncate (Rslt_t &tst_rslt)

This method performs the truncate prior packet test at the nominal overall DCC 0 and 1 bit times.

7.7.2.2.13 Rslt_t decoder_prior (Rslt_t &tst_rslt)

This method performs the variable prior packet test at the nominal overall DCC 0 and 1 bit times.

7.7.2.2.14 Rslt_t decoder_6_byte (Rslt_t &tst_rslt)

This method performs the longer than 6 Byte prior packet test at the nominal overall DCC 0 and 1 bit times.

7.7.2.2.15 Rslt_t decoder_ambig1 (Rslt_t &tst_rslt)

This method performs the 1 ambiguous bit prior packet test at the nominal overall DCC 0 and 1 bit times.

7.7.2.2.16 Rslt_t decoder_ambig2 (Rslt_t &tst_rslt)

This method performs the 2 ambiguous bits prior packet test. The tests are run at the nominal, minimum, and maximum DCC 0 and 1 bit times.

7.7.2.2.17 Rslt_t decoder_str0_ames (Rslt_t &tst_rslt, u_short iclk0t, u_short iclk0h)

This method performs the packet acceptance test at the current overall DCC 0 and 1 bit times with a single DCC 0 bit given by the **`iclk0t`** and **`iclk0h`** times.

7.7.2.2.18 void calc_filler (u_short clk0t, u_short clk1t)

This method calculates the number of Idle packets and command packets necessary to delay by the value given by **`Args.get_fill_msec()`**. This gives the decoder time to respond to a reset, preset, or trigger packet

7.7.2.2.19 Rslt_t send_filler (void)

This method sends the number of Idle packets calculated by the **`calc_filler()`** method.

7.7.2.2.20 const char * err_phrase (bool pre_fail, bool trig_fail, bool rst_fail=false)

This method returns the appropriate error phrase string based on the value of the preset, trigger and optional reset fail status.

7.7.2.2.21 const char * min_phrase (u_short t_min, u_short out_of_range)

This method returns the appropriate minimum DCC 1 bit string.

7.7.2.2.22 const char * max_phrase (u_short t_max, u_short out_of_range)

This method returns the appropriate maximum DCC 1 bit string.

7.7.2.2.23 const char * min_duty_phrase (u_short t_nom, u_short t_min, u_short out_of_range)

This method returns the appropriate minimum duty cycle bit string.

7.7.2.2.24 const char * max_duty_phrase (u_short t_nom, u_short t_max, u_short out_of_range)

This method returns the appropriate maximum duty cycle bit string.

7.7.2.2.25 void print_test_rslt (Rslt_t tst_rslt)

This method prints a message to the console showing whether one or more tests have failed. It is called at the end of each test to give an indication if tests have failed previous to the current test.

7.7.3 New Dec_tst Public Interface

7.7.3.1 The New Dec_tst Public Interface Shall Follow the Current Class as Closely as Possible

7.7.4 Current Dec_tst Protected Implentation

7.7.4.1 Current Protected Implementation Summary

7.7.4.2 Current Protected Implementation Details